**PROJECT #8 – Markov Chain Monte Carlo**

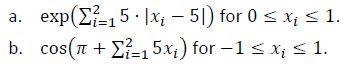
**Divyanshu Sahay  
USC ID: 6364938199  
(dsahay@usc.edu)**

Important Note:

Tool Used: **Python 3.7.4**

1. **Stratified and Importance Sampling**

**Problem Statement:** It is asked to find the Monte Carlo estimates and variances of the given integral in 2 dimensions using a budget of n= 1000 samples. Furthermore, it is asked to implement stratification and importance sampling using the same budget. Following are the expressions:



**Understanding the Problem:** The Monte-Caro estimates for the integral and the variance of the result can be calculated similar to the procedure adopted in the project 4. Again, the methodology relies on the weak law of large numbers where the sample mean converges to population mean for large N.

Further, the second part of the question talks about Stratified and Importance Sampling. Both the methods are used for Variance reduction. Stratified sampling relies on dividing the random variables into stratas i.e. groups and each of these groups are used to sample from the distribution such that the output expression has minimum variance. The strata should define a partition of the population, that is, it should be collectively exhaustive and mutually exclusive: every element in the population must be assigned to one and only one stratum. Then simple random sampling or systematic sampling is applied within each stratum. The objective is to improve the precision of the sample by reducing sampling error. It can produce a weighted mean that has less variability than the arithmetic mean of a simple random sample of the population.

Importance sampling is derived from the notion of inability to sample from probability density, p(x), of the random variable. In this case another density g(x) which satisfies, **g(x)=0 => p(x)=0**, can be used to sample. So,

As mentioned before the function g(x) will be used. The program written will have sufficient comments to explain the algorithms.

**Program Written:**

import random

import math

import numpy as np

import scipy.stats as stats

import statistics

import matplotlib.pyplot as plt

a = -1

b = 1

N  = 1000

result = 0

law = 0

final\_result = []

def func(X,Y):

    k = np.exp(5\*abs(X-5) + 5\*abs(Y-5))

    return k

# def func(X,Y):

#     k = math.cos(math.pi + 5\*X + 5\*Y)

#     return k

for i in range(N):

    total\_samplesX = np.zeros(N)

    total\_samplesY = np.zeros(N)

    law = 0

    for i in range (len(total\_samplesX)):

        total\_samplesX[i] = random.uniform(a,b)

        total\_samplesY[i] = random.uniform(a,b)

    for i in range(N):

        law = law + func(total\_samplesX[i],total\_samplesY[i])

    result = (b-a)\*(b-a)\*(law/N)

    final\_result.append(result)

fig = plt.figure()

plt.hist(final\_result)

print('Variance')

print(statistics.variance(final\_result))

plt.show()

**Stratified and Importance Sampling**

import numpy as np

import math

def func(X,Y):

    k = np.exp(5\*abs(X-5) + 5\*abs(Y-5))

    return k

# def func(X,Y):

#     k = math.cos(math.pi + 5\*X + 5\*Y)

#     return k

N = 10000

fX = np.random.rand(1,N)

fY = np.random.rand(1,N)

X = my\_func(fX,fY)

print('Mean is:', str(np.mean(X)))

print(2\*np.std(X)/np.sqrt(N))

# stratified sampling

K = 20

XSb = np.zeros((K,K))

SS = np.zeros\_like(XSb)

Nij = N/np.power(K,2)

for i in range(0,K):

    for j in range(0,K):

        XS = my\_func((i+np.random.rand(1,int(Nij)))/K,(j+np.random.rand(1,int(Nij)))/K)

        XSb[i][j] = np.mean(XS)

        SS[i][j] = np.var(XS)

SST = np.mean((SS/N))

SSM = np.mean((XSb))

print('Mean with stratified sampling is:', str(SSM))

print(2\*np.sqrt(SST))

# importance sampling

N\_is = 10000

U = np.random.rand(2,N\_is)

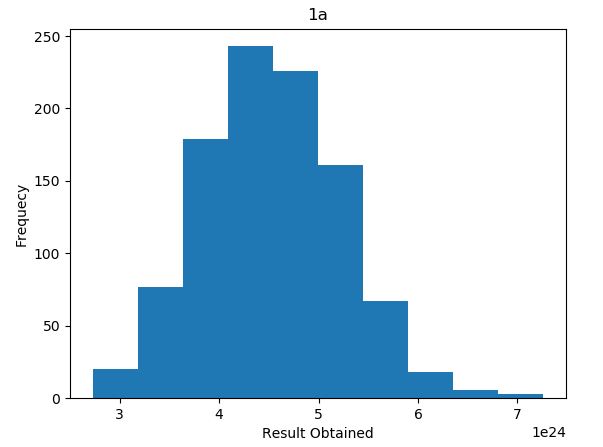
X\_is = np.log(1+(np.exp(1)-1)\*U)

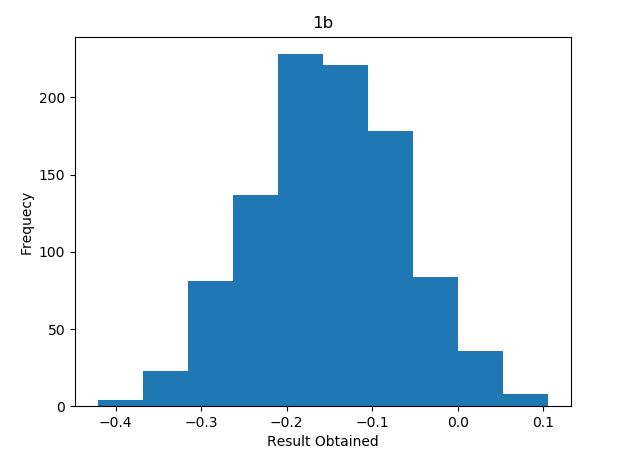
T = np.power((np.exp(1)-1),2)\*np.exp((np.power(np.sum(X\_is,axis=0),2)) - np.sum(X\_is,axis=0))

print('Mean with importance sampling is:',str(np.mean(T)))

print(2\*np.std(T)/np.sqrt(N))

**Results and Observation:**

  
Fig.1. Integral Estimate for 1(a)

  
Fig.2. Integral Estimates (1b)

Output Screen:

Variance

0.007815024398634641

For 1(a)

Mean is: 2.1982909571512694e+20

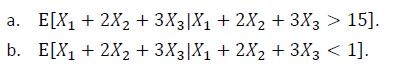
Mean with stratified sampling is: 2.0480493637733492e+20

Mean with importance sampling is: 2.2347452365725684e+20

The outputs are shown above which are obtained through three methods adopted. It be seen that the variance reduction techniques perform better.

1. **Gibb’s Sampling**

**Problem Statement:** It is asked to perform Gibbs Sampling to estimate following:

****

Where, ***Xi*** are independent exponential with mean 1

**Understanding the Problem:** The Gibb’s sampling is the method of sampling from a probability distribution of 2 or more dimensions. It is similar to Metropolis Hastings with a major difference that unlike Metropolis Hasting we accept all proposals. The most important fact that must be understood is that, Gibbs sampling relies on knowing the conditional distribution and being able to sample from it.

In the given problems we have 3D systems with a conditional distribution that governs the sampling. The approach in Gibb’s Sampling would be to make an initial relevant guess for two of the random variables and then sample the other based on the given condition. This process is repeated for all the three variables and they are updated on each iteration.

**Program Written 1(a):**

import numpy as np

import random

import matplotlib.pyplot as plt

X1\_list = []

X2\_list = []

X3\_list = []

X\_sum\_list = []

# part A

# lets choose starting point for the simulation as follows:

X1 = 5

X2 = 7

for i in range(1000):

    # as we have restricted our sampling now X3 can only be slected strcitly less than 3

    X3 = random.expovariate(1) # we make an initial sample and then see if we can accept it

    while(not 3\*X3 > (15-(X1+2\*X2))):

        X3 = random.expovariate(1) # we keep on sampling unless we achieve our desired condition.

    X3\_list.append(X3)

    X\_sum\_list.append(X1 + 2\*X2 + 3\*X3)

    X2 = random.expovariate(1)

    while(not 2\*X2 > (15-(X1+3\*X3))):

        X2 = random.expovariate(1)

    X2\_list.append(X2)

    X\_sum\_list.append(X1 + 2\*X2 + 3\*X3)

    X1 = random.expovariate(1)

    while(not X1 > (15-(2\*X2+3\*X3))):

        X1 = random.expovariate(1)

    X1\_list.append(X1)

    X\_sum\_list.append(X1 + 2\*X2 + 3\*X3)

print('Expected value of the desired sum')

print(sum(X\_sum\_list)/len(X\_sum\_list))

plt.plot(X\_sum\_list)

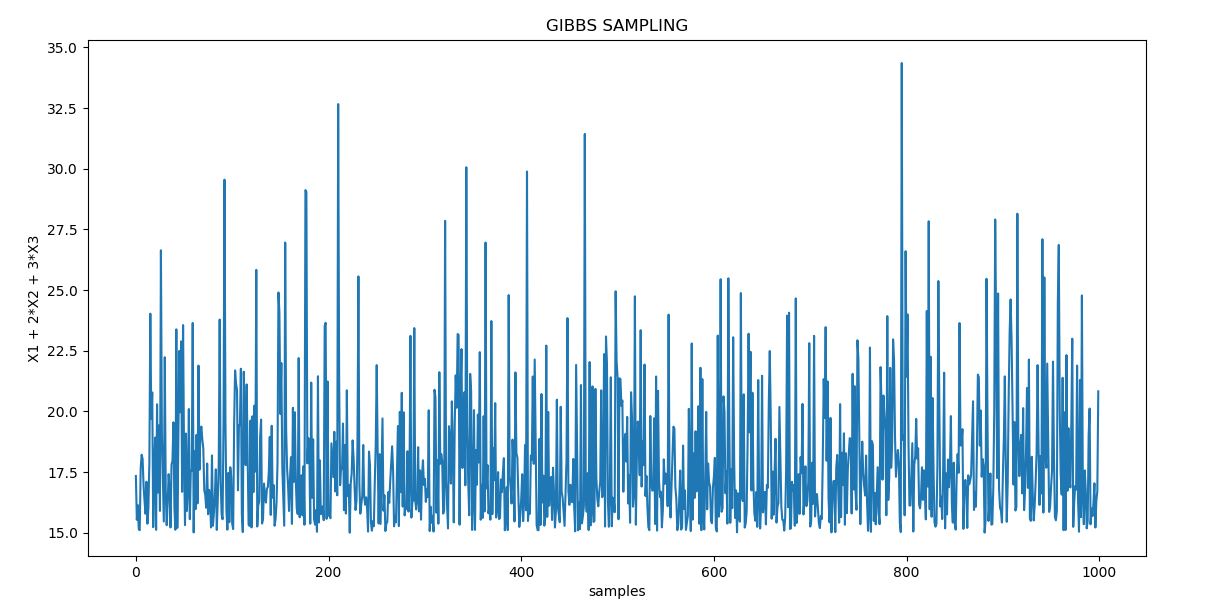
plt.ylabel('X1 + 2\*X2 + 3\*X3')

plt.xlabel('samples')

plt.title('GIBBS SAMPLING')

plt.show()

**Results and Observation:**

**  
Fig. 3. 2(a) Gibb’s Sampling**

Expected value of the desired sum

0.7231563252328108

It can clearly be seen that as required all the samples are greater than 15. This is in congruence with our expectation. Also, the efficiency is very high. The expected values of the desired sum is also shown.

**Program Written 1(b):**

import numpy as np

import random

import matplotlib.pyplot as plt

X1\_list = []

X2\_list = []

X3\_list = []

X\_sum\_list = []

# part A

# lets choose starting point for the simulation as follows:

X1 = 0.1

X2 = 0.2

# as we have restricted our sampling now X3 can only be slected strcitly less than 3

for i in range(1000):

    X3 = random.expovariate(1)

    while(not 3\*X3 < (1 - (X1+2\*X2))):

        X3 = random.expovariate(1)

    X3\_list.append(X3)

    X\_sum\_list.append(X1 + 2\*X2 + 3\*X3)

    X2 = random.expovariate(1)

    while(not 2\*X2 < (1 - (X1+3\*X3))):

        X2 = random.expovariate(1)

    X2\_list.append(X2)

    X\_sum\_list.append(X1 + 2\*X2 + 3\*X3)

    X1 = random.expovariate(1)

    while(not X1 < 1 - (2\*X2+3\*X3)):

        X1 = random.expovariate(1)

    X1\_list.append(X1)

    X\_sum\_list.append(X1 + 2\*X2 + 3\*X3)

plt.plot(X\_sum\_list)

print('Expected value of the desired sum')

print(sum(X\_sum\_list)/len(X\_sum\_list))

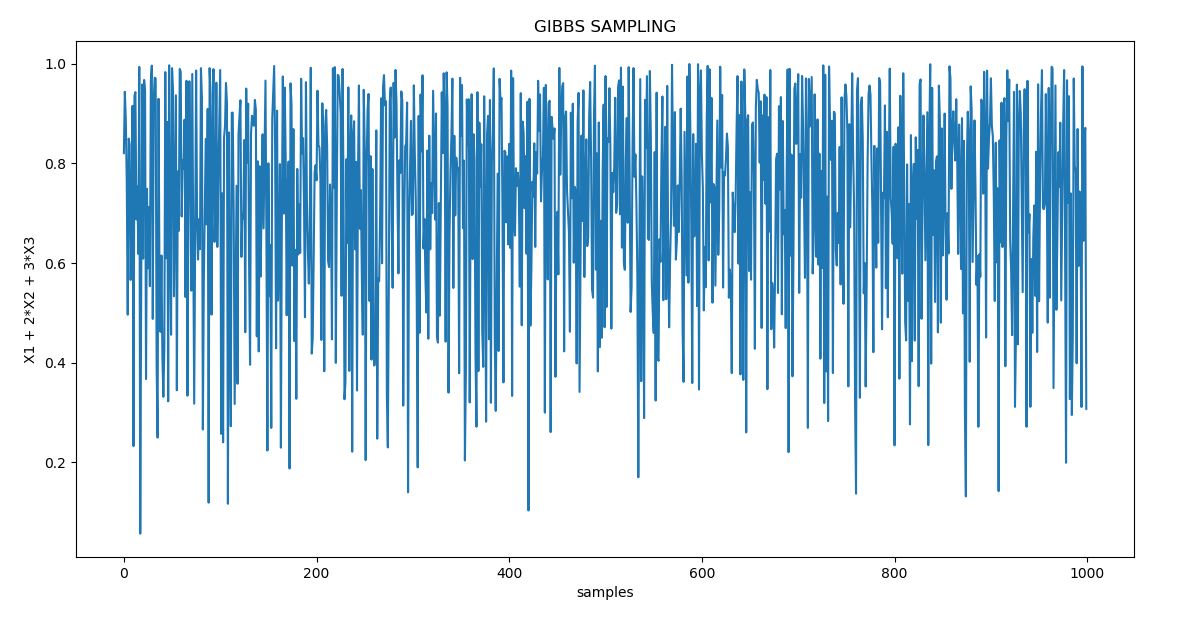
plt.ylabel('X1 + 2\*X2 + 3\*X3')

plt.xlabel('samples')

plt.title('GIBBS SAMPLING')

plt.show()

**Results and Observation:**

  
**Fig. 4. 2(b) Gibb’s Sampling**

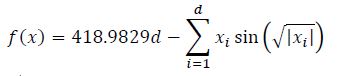
Expected value of the desired sum

17.993867094754272

Again, the Gibbs sampling appears to be pretty efficient and the expected value also is in direct congruence of our expectation.

1. **Simulated Annealing to find the Global Minima:**

**Problem Statement:** In this problem we need to find the global minima for the Schwefel Function given as follows:



Following are the requirements:

1. A contour plot of 2D Schwefel Surface
2. Find Global minima using simulated annealing (beginning at (0,0))
3. Comparing the simulation behaviour for: Exponential, Polynomial and Logarithmic cooling schedules.
4. The algorithm must be run for different iteration counts: N = 50, 200, 1000, 10000, and a histogram must be generated for each case showing global minima.
5. The 2D sample path of the best estimate must be overlayed with 2D contour plot.

**Understanding the Problem:** This problem and the last problem introduces Simulated Annealing. Wikipedia says: *The name and inspiration come from annealing in metallurgy, a technique involving heating and controlled cooling of a material to increase the size of its crystals and reduce their defects.* *This notion of slow cooling implemented in the simulated annealing algorithm is interpreted as a slow decrease in the probability of accepting worse solutions as the solution space is explored. Accepting worse solutions is a fundamental property of metaheuristics because it allows for a more extensive search for the global optimal solution*.

Thus adopting this approach, we can try to find the global minima iteratively by starting from a solution and selecting the next solution based on its quality. In the process of selecting new solutions we also accept some bad/worse so that we are able to browse through the entire solution space and don’t get stuck at some minima. In the context of problem 3, the algorithm is explained in comments of the program written:

**Program Written:**

import numpy as np

import matplotlib

import matplotlib.pyplot as plt

import scipy.stats as stats

from mpl\_toolkits.mplot3d import Axes3D

from matplotlib import cm

from matplotlib.ticker import LinearLocator, FormatStrFormatter

N\_r = 500  #as per the limit of the hyper cube

x = np.linspace(-N\_r,N\_r,100)

y = np.linspace(-N\_r,N\_r,100)

X, Y = np.meshgrid(x, y)

z1 = X\*np.sin(np.sqrt(np.abs(X)))

z2 = Y\*np.sin(np.sqrt(np.abs(Y)))

Z = 418.9829\*2 - z1 -z2

plt.figure(num=None,dpi=100)

plt.contourf(X,Y,Z) # plotting the contour

plt.title('Contour Plot')

plt.colorbar()

plt.show()

cplot = plt.figure(num=None,dpi=150)

ax = cplot.add\_subplot(111,projection='3d')

surf = ax.plot\_surface(X,Y,Z,cmap=cm.coolwarm)

cbar = cplot.colorbar(surf, shrink=0.5, aspect=5)

cbar.minorticks\_on()

plt.show()

def sch\_fun(x1,x2):

    return (418.9829\*2 -  x1\*np.sin(np.sqrt(np.abs(x1))) -  x2\*np.sin(np.sqrt(np.abs(x2))))

# simulated annealing to find the minimum

N = 10000

x1 = 0 #we begin our simulation at origin

x2 = 0

x1\_list = []

x2\_list = []

T = 100

value\_list = []

value\_list.append(sch\_fun(x1,x2))

for i in range (1,N):

    x1\_prop = x1 + np.random.normal(0,50) # creating a new propsal solution

    x2\_prop = x2 + np.random.normal(0,50)

    alpha = np.exp((sch\_fun(x1\_prop,x2\_prop) - sch\_fun(x1,x2))/T) #the threshold value

    if (sch\_fun(x1\_prop,x2\_prop) < sch\_fun(x1,x2)): # cheking the coondition

        x1 = x1\_prop #updating new va

        x2 = x2\_prop

    else:

        if (np.random.rand() < alpha): #now checking with the alpha, where we sometimes accpet bad values

            x1 = x1\_prop

            x2 = x2\_prop

    #T = 100/(np.log(i+2)) #logarithmic cooling function

    #T = 100 - (100/N)\*(i+1) #polynomial cooling function

    T = 100\*np.exp(-(100/N)\*np.sqrt(i+1)) #exponenential cooling function

    x1\_list.append(x1)

    x2\_list.append(x2)

    value\_list.append(sch\_fun(x1,x2))

plt.hist(value\_list)

plt.show()

plt.plot(x1\_list, x2\_list, 'C3', zorder=1, lw=3)

plt.scatter(x1\_list, x2\_list, s=1, zorder=2)

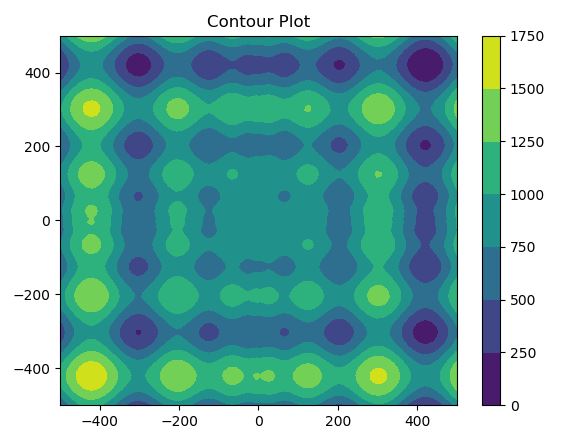
plt.contour(X,Y,Z)

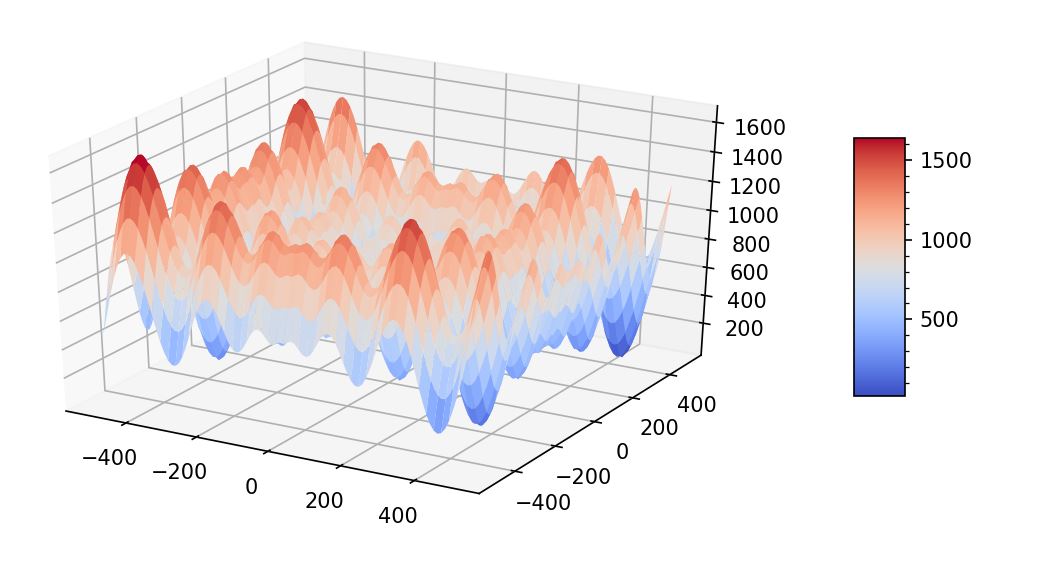
plt.title('final path')

plt.tight\_layout()

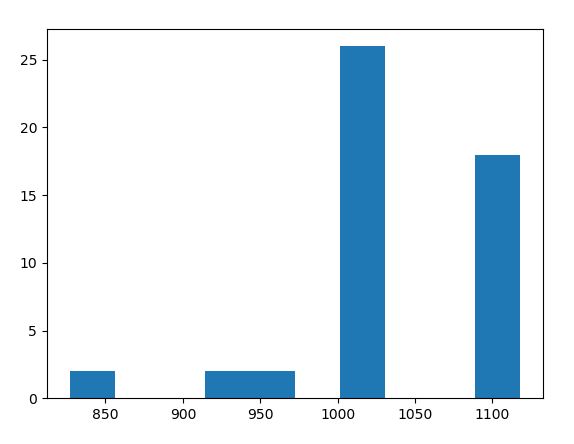
plt.show()

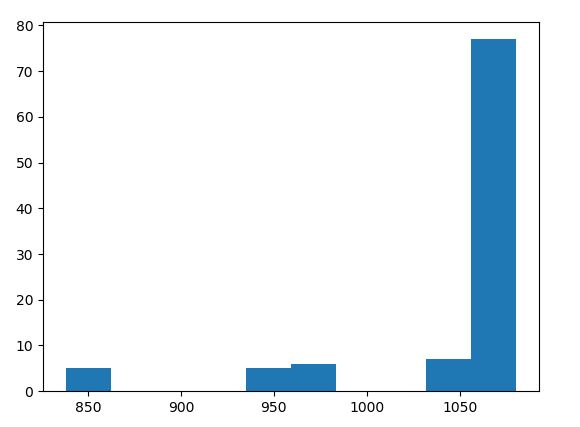
**Results and Observations:**

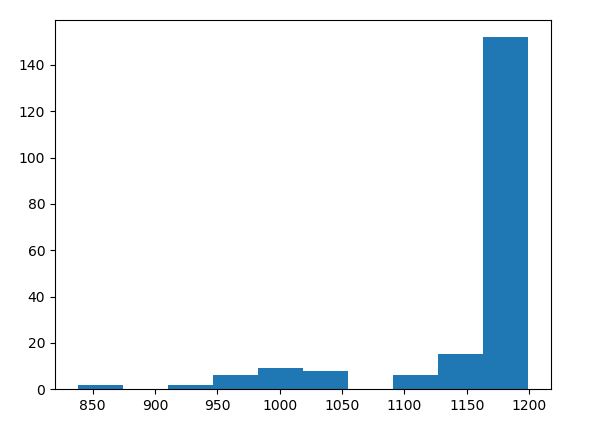
**  
Fig.5. Contour Plot**

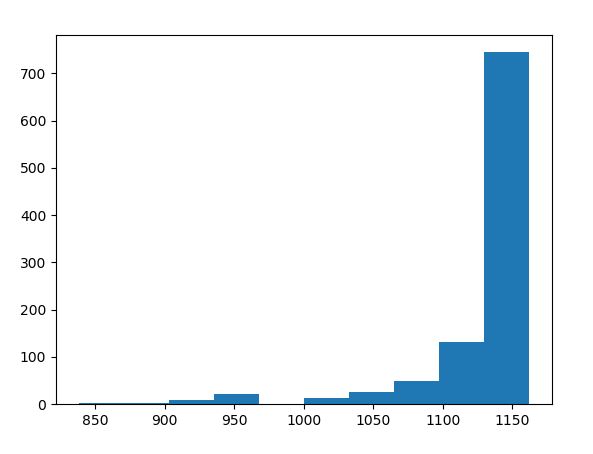
**  
Fig.6 Surface Plot**

Now the histograms are generated for each N values,

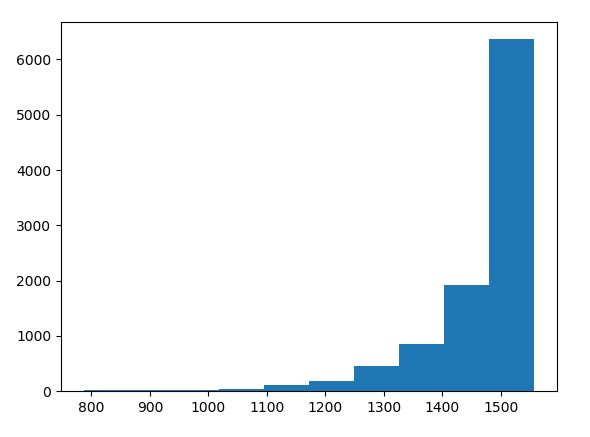
  
**Fig.7 N = 50**

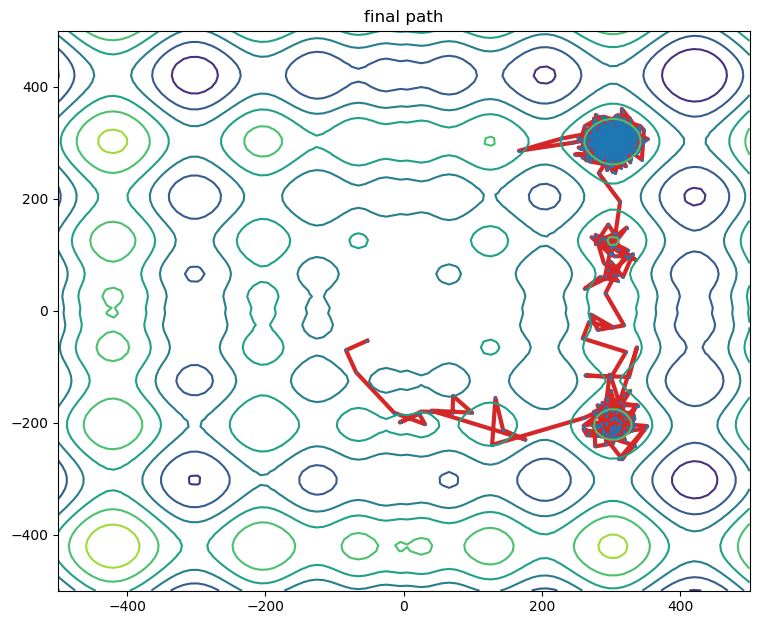
  
**Fig.8.N = 100**

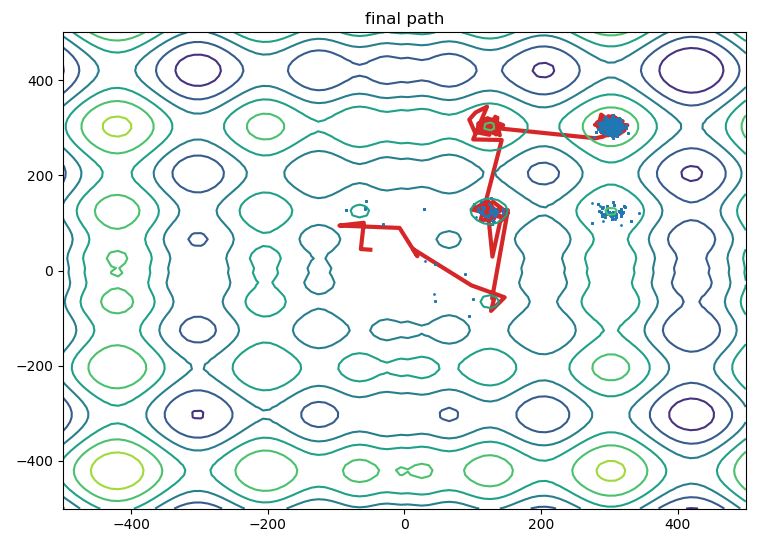
  
**Fig.9.N = 200**



**Fig.10 N = 1000**

  
**Fig. 11.N = 10000**

  
**Fig.12. Contour Plot overlayed with Global Minimum (iteration 2)**

  
**Fig.13 Contour Plot overlayed with Global Minimum (iteration 1)**

These plots were obtained with logarithmic cooling routines. It can be seen in the contour plot that the global minimum is getting reached. I have taken snap of two such iterations.

1. **Dual of a Travelling Salesman Problem:**

**Problem Statement:** The problem talks about a road trip to all states of United States of America (excluding Alaska and Hawaii) using the most optimum/smallest path found by Simulated Annealing. Following are the requirements:

1. Estimate minimal path using Simulated Annealing
2. Plot the best path on the X-Y axis
3. Plot total tour distance as a function of time.
4. Comment on the estimation rate required.

**Understanding the Problem:** This is similar to what we did in problem 3. Now, it is asked to find the optimum path. The approach behind the algorithm is as follows: We start with a particular location and generate a random path by choosing cities randomly. Thereafter, we keep on swapping two cities chosen randomly to see if the path length becomes better. Similar to what we did in problem 3, we don’t always reject a bad prospect, thereby opening us to the option of exploring the entire workspace. Finally, when we repeat this process for a long time we end up getting the most optimum solution. This is one of the proposed solution of a NP-Hard problem which is obviously not the most optimum. This can give a good solution indeed. The exact algorithm will be explained in the code through comments.

Also, the code written for this is an adaptation of the code provided by the professor in the tutorial.

**Program Written:**

import numpy as np

from sklearn.metrics import pairwise\_distances

from sklearn.utils.random import sample\_without\_replacement

import numpy as np

import matplotlib.pyplot as plt

from scipy.spatial import distance

cities\_coord = []

with open('D:\\SHIVAM\\USC\_STUDY\\EE511\\project8\\code\\Q4\\xy\_cord.txt') as f:

    for line in f:

        total = line.split()

        cities\_coord.append([float(total[0]),float(total[1])])

    # x1 = [float(line.split()[0]) for line in f]

print(cities\_coord[3])

# echanged the 1st and 4th row to start from sacramanto

N\_cities = 48

p\_len = 0

for a1 in range(0,N\_cities-1):

    p\_len = p\_len + distance.euclidean(cities\_coord[a1],cities\_coord[a1+1])

print('Initial path length:',str(p\_len))

# lets name this 1st path as path where the cities as given in file are nubered from 0 to 47

path = np.arange(0,N\_cities)

num\_iter = 1000

# Save the paths and lengths

pathHistory = np.zeros((num\_iter,N\_cities))

lenHistory = []

thresh\_ar = []

# plot cities and initial path

x\_coord = []

y\_coord = []

plt.figure()

for i in range(len(cities\_coord)):

    x\_coord.append(cities\_coord[i][0])

    y\_coord.append(cities\_coord[i][1])

plt.plot(x\_coord, y\_coord, 'C3', zorder=1, lw=3)

plt.scatter(x\_coord, y\_coord, s=120, zorder=2)

plt.title('Initial path')

plt.tight\_layout()

plt.show()

# now we will chooswe new paths by swapping two cities selected randomly

iter\_count = 1

path\_new = []

c = 1

while iter\_count < num\_iter:

    iter\_count = iter\_count + 1

    # Create path p2 by randomly swap two cities

    # index of two cities for the new path

    swap\_i, swap\_j = np.random.choice(N\_cities, 2)

    path\_new = np.copy(path)

    path\_new[swap\_i], path\_new[swap\_j] = path\_new[swap\_j], path\_new[swap\_i]

    # initialize new path length

    p\_len2 = 0

    for a1 in range(0,N\_cities-1):

        p\_len2 = p\_len2 + distance.euclidean(cities\_coord[path\_new[a1]],cities\_coord[path\_new[a1+1]])

    thresh = np.power((1+iter\_count),((p\_len - p\_len2)/c))

    if p\_len2 - p\_len <= 0:

        path = np.copy(path\_new)

        p\_len = np.copy(p\_len2)

    else:

        if np.random.rand() <= thresh:

            path = np.copy(path\_new)

            p\_len = np.copy(p\_len2)

          # bookeeping

    pathHistory[iter\_count-1][0:len(path\_new)] = path\_new

    lenHistory.append(p\_len2)

    thresh\_ar.append(thresh)

plt.figure(num=None,dpi=100)

plt.plot(lenHistory)

print(p\_len)

plt.title('Length of path in each iteration')

plt.show()

x\_coord\_f = []

y\_coord\_f = []

ind\_f = pathHistory[-1,:].astype(int)

print(ind\_f)

for i in range(len(ind\_f)):

    x\_coord\_f.append(cities\_coord[ind\_f[i]][0])

    y\_coord\_f.append(cities\_coord[ind\_f[i]][1])

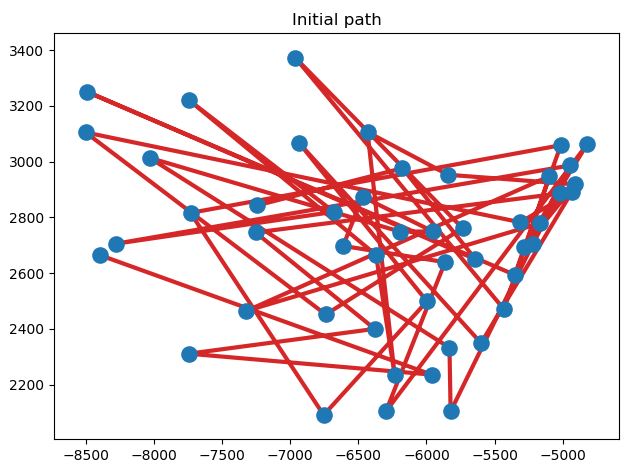
plt.figure(num=None,dpi=100)

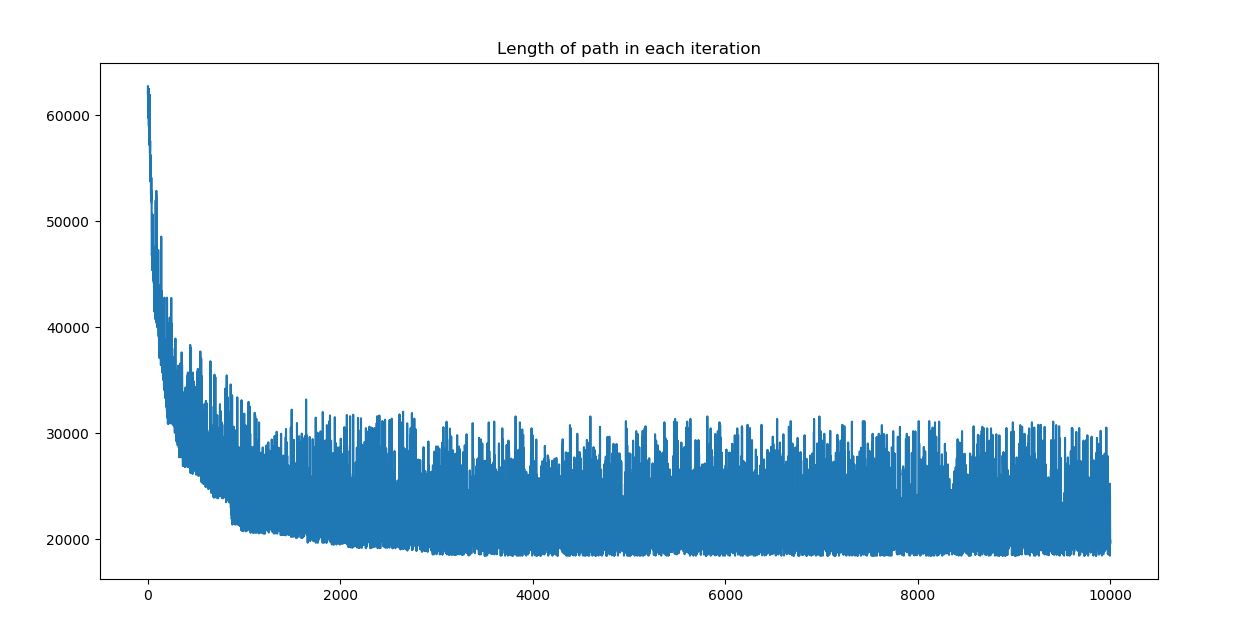
plt.title('Final path')

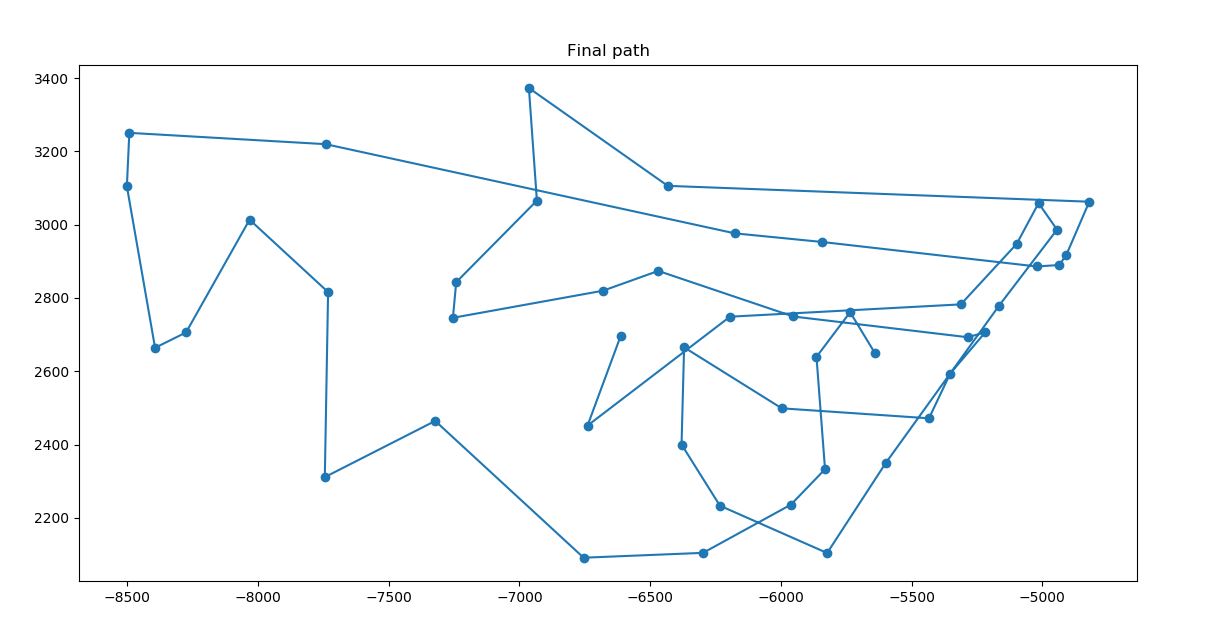
plt.plot(x\_coord\_f, y\_coord\_f, '-o')

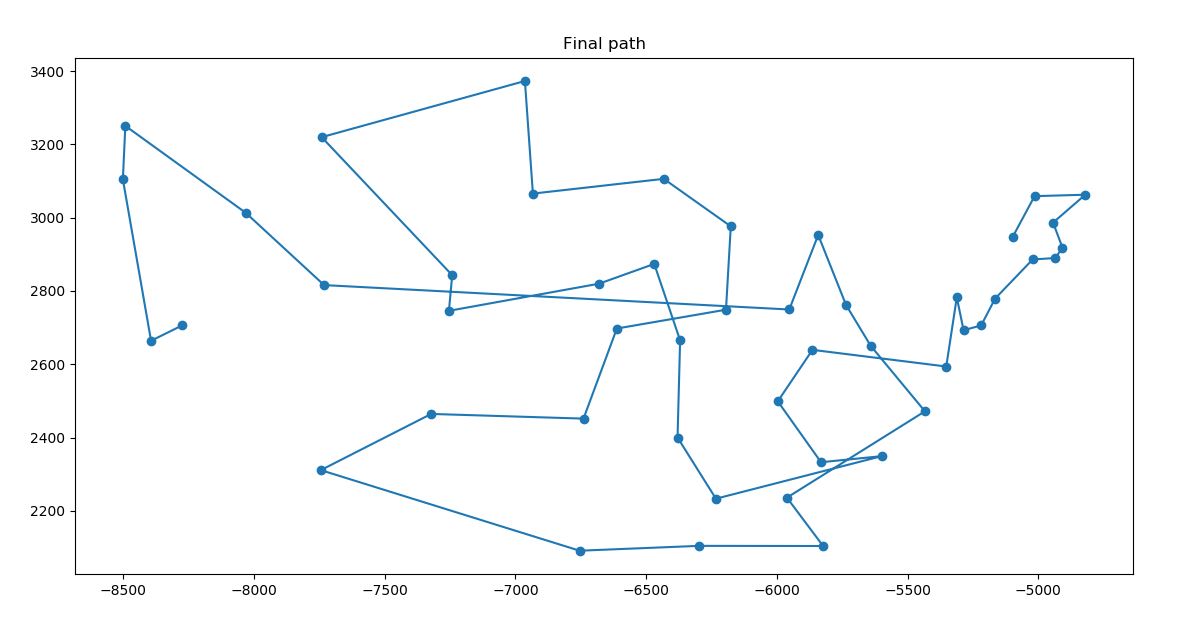
plt.show()

**Results and Observation:**

**  
Fig.14. The Initial path selected Randomly**

**  
Fig.15. Length of path in each iteration.**

**  
Fig.16 The Final Path (iteration 1).**

**  
Fig.17 The Final Path (iteration 2).**

**Output Screen:**

Initial path length: 61665.50179497673

Optimum path length: 15501.177232849048

10,000 iterations were required to obtained the above shown graph. It can be seen that the randomness as seen in the first estimate is definitely reduced but still this cannot be said to be the most optimum. Theoretically speaking, if the iterations are run for infinite number of times, we will get the most optimum solution of this problem. Finally, it can be how drastically the path length has reduced.